A Software Simulation Study of the Long Constraint Length VLSI Viterbi Decoder

S. Arnold and F. Pollara
Communications Systems Research Section

A software simulation of long constraint length Viterbi decoders has been developed. This software closely follows the hardware architecture that has been chosen for the VLSI implementation. The program is used to validate the design of the decoder and to generate test vectors for the VLSI circuits.

I. Introduction

Convolutional codes have been used on deep space probes for several years. During the last few years, TDA Advanced Systems undertook a research effort [1] to develop advanced coding techniques capable of gaining an additional 2 dB over the present performance of deep space missions. Current coding systems are based on a $K = 7$, $r = 1/2$ convolutional code concatenated with an 8-bit (255,223) Reed–Solomon (RS) code, where K is the constraint length and r is the code rate.

The main result of this research effort was the discovery of new convolutional codes with $K = 15$ and $r = 1/6$ which exceed the 2-dB goal when concatenated with a 10-bit (1023,959) RS code. Recently, the delay imposed on the Galileo mission introduced the possibility of including a $K = 15$, $r = 1/4$ code in this mission. This experimental code [2] will gain approximately 1.5 dB over the current NASA-standard code. The Galileo experiment, together with the potential offered by these coding gains for future missions, has led to an effort to build a VLSI-based Viterbi decoder capable of decoding codes with K up to 15 and $r = 1/n$, $n = 2,3,4,5,6$, at speeds approaching 1 Mbit/s.[1]

II. Decoder Architecture

The complexity of a Viterbi decoder depends mainly on the constraint length K, since the number of states is 2^{K-1}. The decoder for the new $K = 15$ codes is approximately 256 times more complex than the current MCD (Maximum-likelihood Convolutional Decoder) used in the DSN stations to decode $K = 7$ codes. The requirement on the information data rate forces the use of heavily parallel architectures.

After evaluation of several design alternatives, it was decided to use a fully parallel architecture consisting of $2^{K-2} = 8192$ physical butterflies operating in parallel. Each butterfly uses bit-serial arithmetic to perform the internal operations of add–

[1] J. Statman, "Preliminary Design Review for Big Viterbi Decoder (BVD)," JPL internal document, March 31, 1988.

compare–select, since this is more suitable to fast VLSI circuits, and represents the metrics as 16-bit numbers. Each butterfly contains two states of the decoder and outputs two decision bits to the trace-back memory. The 8192 butterflies are organized in identical VLSI chips containing 32 butterflies each, and in 16 identical boards containing 16 chips each [3].

The concern was to develop a software simulation of the complete decoder so that (1) several new design ideas could be tested and validated; and (2) test vectors could be generated for signals at various key points in the decoder and then used to test the VLSI design. Given the complexity and the cost of this project, it was necessary to have a complete software decoder that closely emulated the hardware architecture and demonstrated the validity of the design.

III. Software Decoder

The software decoder consists of a program developed on a SUN 3/260 workstation and written in C-language. Since the program runs on a sequential computer, it scans through the butterflies in sequential order, while the hardware performs all these operations in parallel. The decoder is based on the hardware design summarized in Figs. 1 and 2.

Figure 1 represents the *metric computer* module present in each butterfly. It takes the received symbols in sign and magnitude representation and computes the two branch metrics, p and q, as two 16-bit numbers. The register denoted as $LABEL_i$ is initialized at startup time and contains an appropriate label for the ith butterfly. The value of this label is provided by the module *encoder*, whose operation is described by the flow diagram of Fig. 3. Here NB represents the total number of butterflies (8192), i is the index of the current butterfly, and j the index of encoded symbols e_j. First, the n encoded symbols e_j are computed for the current butterfly. Then $LABEL_i$ is just given by the decimal equivalent of the binary array $(e_0, e_1, \ldots, e_5)$. The other input to the metric computer module, r_{max}, is just the sum of the magnitudes of the received symbols for each information bit time. Notice that the diagram in Fig. 1 shows six input received symbols, but it can be used for any code rate $r = 1/n$, $n = 2,3,4,5,6$, by setting the unused symbols to zero. Figure 4 shows a flow diagram representing the computations taking place in the software. The variable j counts the received symbols modulo n.

The *add–compare–select* circuit of Fig. 2 takes the branch metrics p and q just computed and the previously computed accumulated metrics m_{i0} and m_{i1} from states $i0$ and $i1$ and generates the updated metrics m_{j0}, m_{j1} and the decision bits bit_0 and bit_1, which are stored in the trace-back memory. This memory is organized in three banks of L bits each, where L is the path truncation length. Decoded bits are given by the trace-back performed on the bank containing the "oldest" decision bits. The detailed operation of the add-compare–select module is shown in the flow diagram of Fig. 5. The test for overflow is performed on the output accumulated metric m_{j0} of butterfly number zero. Renormalization occurs if the two most significant bits of m_{j0} are both equal to one. In this case, the most significant bit of all accumulated metrics is reset to zero to prevent overflow of the metrics. The decoder described in this article and its future VLSI implementation can decode any code with connection vectors $G_i = (x_{i0}, x_{i1}, \ldots, x_{i14})$, where $x_{ij} \in (0,1)$ and $x_{i0} = x_{i14} = 1$. Code search results [1], [2] show that good codes always meet the constraint of having a leading and trailing "1" in the connection vectors. Because of this constraint, only two branch metrics, p and q, need to be computed. When $K < 15$, this constraint is no longer met, but it can be observed that in this case m_{j1} is always equal to m_{j0}, as shown in [3]. This is accomplished with the switch in Fig. 2 or the test $(K < 15)$ in Fig. 5.

IV. Operation of the Software Decoder

Testing a large Viterbi decoder is a complex task, since some programming errors may be revealed only by particular input sequences or error patterns. This decoder has been tested first against an existing software decoder for $K = 7$ and $r = 1/2$, which has been extensively used in the past. After it was ascertained that the two programs had identical behavior, the new program was tested with various other codes, and it also performed according to expectations.

To run the program, which is reproduced in the Appendix, the user must enter K, the inverse n of the rate, the path truncation length L, and the generator polynomials in octal. Also, the names of the files used to get the input received symbols and to write the decoded bits must be provided. Currently, the output consists of the decoded information bits and is written to a disk file. The test signals to be used for future testing of the VLSI circuits can be obtained by inserting print statements anywhere desired in the program.

References

[1] J. H. Yuen and Q. D. Vo, "In Search of a 2-dB Coding Gain," *TDA Progress Report 42-83*, vol. July–September 1985, Jet Propulsion Laboratory, Pasadena, California, pp. 26–33, November 15, 1985.

[2] S. Dolinar, "A New Code for Galileo," *TDA Progress Report 42-93*, vol. January–March 1988, Jet Propulsion Laboratory, Pasadena, California, pp. 83–96, May 15, 1988.

[3] O. Collins, "Techniques for Long Constraint Length Viterbi Decoders," Intern. Symposium on Information Theory, Kobe, Japan, p. 28, June 1988.

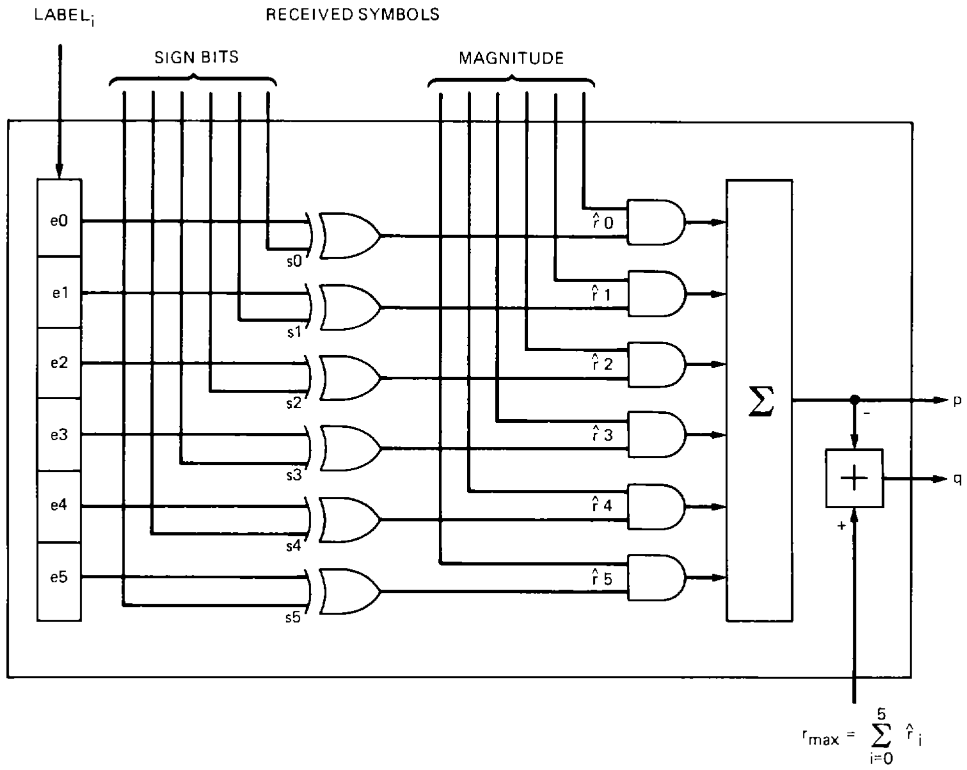
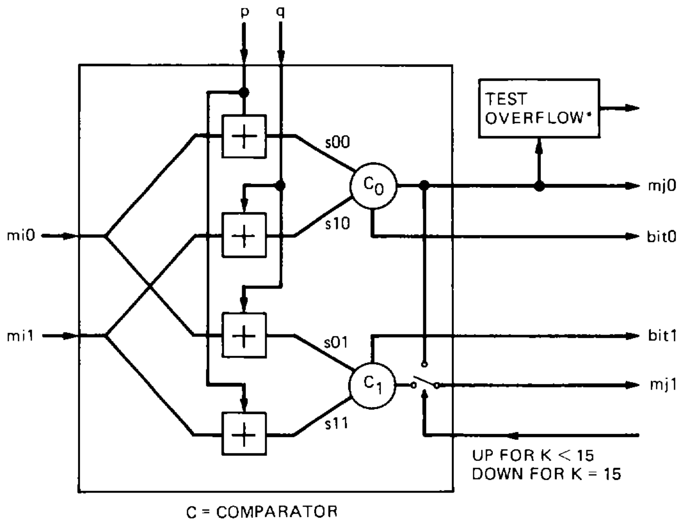


Fig. 1. Metric computer hardware diagram



*THIS IS USED ONLY FOR BUTTERFLY = 0

Fig. 2. Add–compare–select hardware diagram

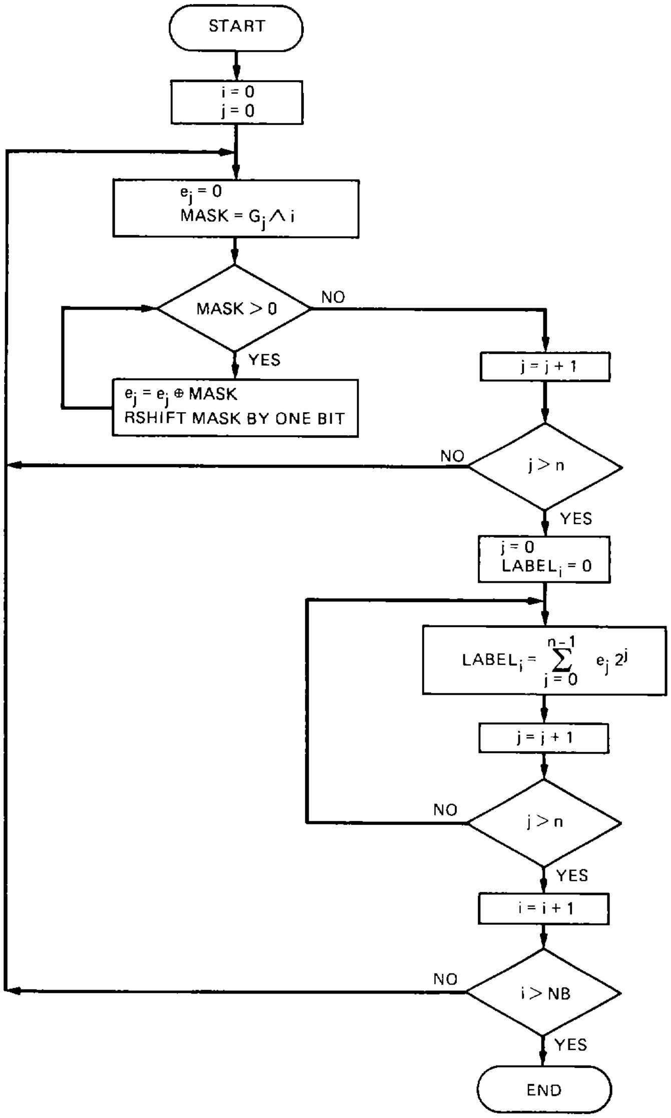


Fig. 3. Encoder flow diagram

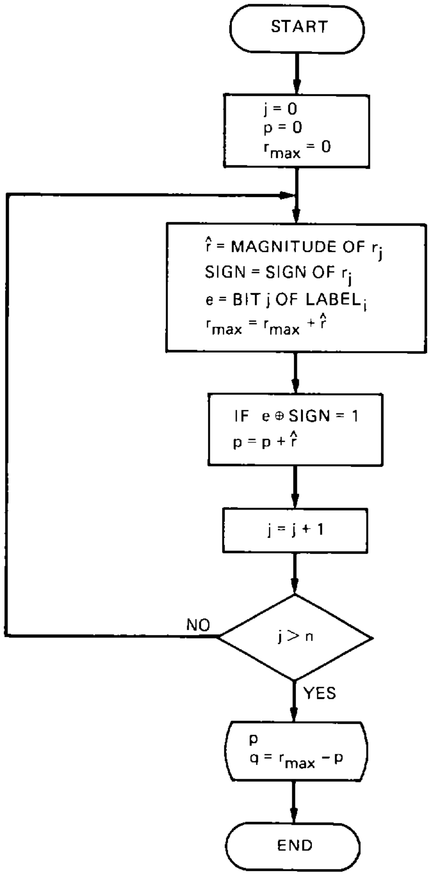


Fig. 4. Metric computer flow diagram

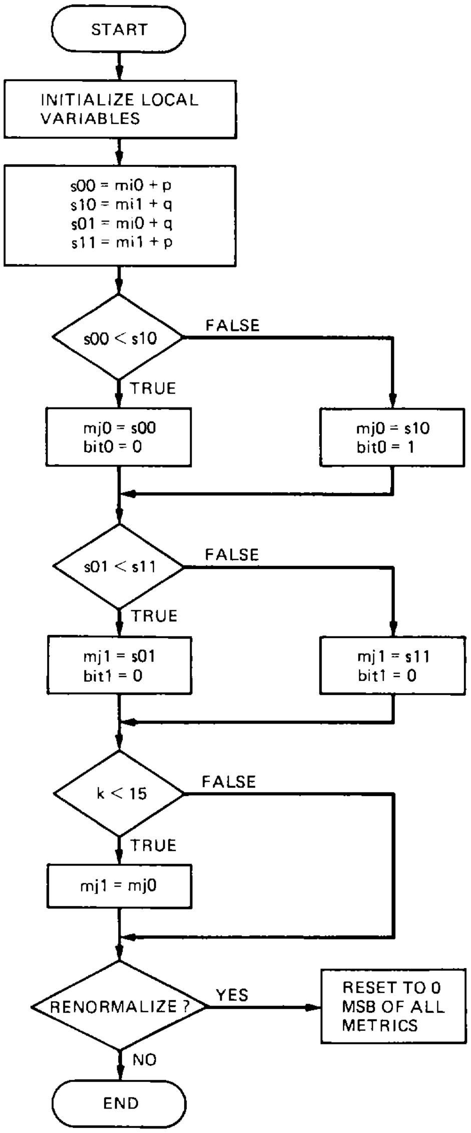


Fig. 5. Add–compare–select flow diagram

Appendix

```
# include <stdio.h>

/*********************************** VLSI ***********************************/
/*                                                                          */
/*  This program simulates the long constraint length VLSI Viterbi decoder. */
/*  It allows the user to decode convolutional codes with constraint length */
/*  up to 15 and code rate 1/2 to 1/6.                                      */
/*                                                                          */
/****************************************************************************/

int n;                         /* rate= 1/n */
int L;                         /* buffer length */
int p,q;                       /* branch metrics */
int n_tb;                      /* traceback addresses */
int time;                      /* traceback time */
int butt;                      /* loop counter */
int NS;                        /* number of states */
int NB;                        /* number of butterflies */
int k;                         /* constraint length */
int dec;                       /* decoding bank */
int mi0, mi1, mj0, mj1;        /* accumulated metrics */
int blk_time;                  /* time in traceback */
int bit_no;                    /* number of symbols decoded */
int GP[6];                     /* generator polynomials */
int out[100];                  /* temp storage for decoded bits */
int outr[100];                 /* storage for decoded bits */
int LABEL[8192];               /* butterfly labels */
int metric[16384], old_metric[16384];      /* accumulated metric storage */
char flag;                     /* renormalization flag */
char bit0, bit1;               /* decision bits */
char RAM[16384][100][3];       /* traceback RAM*/

main ()
{
        int tb;                        /* traceback (tb) bank */
        int Mo;                        /* parameter to calculate memory size */
        int bank;                      /* loop counter */
        int blk_no;                    /* number of blocks decoded */
        int blk_par;                   /* block parity (0 or 1) */
        int n_dec;                     /* addresses of decoded bits */
        int state;                     /* loop counter */
        int symbol_no;                 /* loop counter */
        int state0, state1;            /* current states of butterfly */
        int prev_state0, prev_state1;     /* previous states of butterfly */
        int recsym[6];                 /* received symbols (8-bit) */
        char decinp[10], decout[10];      /* input and output files of decoder */
        FILE *fp1, *fp2;                    /* input/output file pointers */

        printf ("The simulation can decode binary data with a constraint length");
        printf ("k < = 15 and code rate of 1/2 to 1/6");

        printf ("Enter constraint length k");
        scanf ("%d",&k);
        printf ("Enter number of symbols n (2-6)");
        scanf ("%d",&n);
        printf ("Enter length of traceback buffer L");
        scanf ("%d",&L);

        for (symbol_no= 0; symbol_no< n; symbol_no+ + ) {
                printf ("Enter generating polynomial GP[%d] in OCTAL > ",symbol_no);
                scanf ("%o",&GP[symbol_no]);
                if (k < 15) GP[symbol_no] < < = (15 - k);
        }

        printf ("Enter binary input filename");
        scanf ("%s",decinp);
        printf ("Enter output filename that will contain decoded bits");
        scanf ("%s",decout);

        fp1 = fopen (decinp,"r");           /* open file of received symbols */
        fp2 = fopen (decout,"w");           /* open file for decoder output */

        bit_no = 0;                         /* set bit counter to zero */
        symbol_no = 0;                          /* set symbol counter to zero */
        flag = 0;                           /* set renormalization flag to zero */
        n_tb = 0;                           /* set starting tb addr. to zero */

        Mo = 14;
        NS = 01 < < Mo;                         /* number of states */
        NB = NS/2;                          /* number of butterflies */

/* set storage of decoded bits to zero */

        for time = 0; time < L; time+ + ) out[time] = 0;

/* initialize metrics, accumulated metrics, and traceback RAM to zero*/

        for (state = 0; state < NS; state+ + ) {
                metric[state] = 0;
                old_metric[state]  = 0;
                for (time = 0; time < L; time+ + )
                        for (bank = 0; bank < 3; bank+ + )
                                RAM[state][time][bank] = 0;
        }

/* generate the labels that are assigned to the butterfly */

        encoder( );

/* receive data bits and enter decoder loop */

        while ((recsym[symbol_no] = getc (fp1)) != EOF) {
                symbol_no+ + ;

                if (symbol_no = = n) {
                        symbol_no = 0;
```

```
/* check value of flag to determine to renormalize accumulated metrics */

                              if (flag = = 0)
                                     for (state = 0; state < NS; state+ + )
                                          old_metric[state] = metric[state];
                              else {
                                     for (state = 0; state < NS; state+ + )
                                          old_metric[state] = metric[state]&077777; /* clear MSB */
                                     flag = 0;
                              }

                              blk_time = bit_no%L;

/* check to see if new traceback must be started */

                              if (blk_time = = 0) {
                                   blk_no = bit_no/L;
                                   blk_par = blk_no%2;
                                   tb = blk_no%3;
                                   dec = (tb+ 1)%3;
                                   n_dec = n_tb;
                                   n_tb = 0;
                                   for (time = 0; time < L; time+ + ) outr[time] = out[time];
                              }

/* determine whether to move left or right through traceback memory */

                              if (blk_par = = 0) time = L - blk_time - 1;
                              else          time = blk_time;

/* generate the addresses for the decoded bits and traceback */

                              n_dec = (n_dec > > 1) |(NB*RAM[n_dec][time][dec]);
                              n_tb = (n_tb > > 1) |(NB*RAM[n_tb][time][tb]);

                              out[blk_time] = (n_dec > > 5)&01;    /* extract decoded bits */

/* Generate branch metrics associated with new received symbol, add to  */
/* existing accumulated metrics, determine smallest accumulated metric  */
/* at current state, and output decision bits to traceback memory       */

                              for (butt = 0; butt < NB; butt+ + ) {

/* compute the two current states of butterfly and their associated previous states */

                                   state0 = butt < < 1;
                                   state1 = state0 + 1;
                                   prev_state0 = butt;
                                   prev_state1 = prev_state0 |NB;

                                   metric_comp(recsym);          /* call metric computer */

                                   mi0 = old_metric[prev_state0];
                                   mi1 = old_metric[prev_state1];
```

```
                                        add_comp_select();          /* call add, compare, and select */

                                        metric[state0] = mj0 ;
                                        metric[state1] = mj1 ;

/* write to traceback RAM, the bits at corresponding state of butterfly */

                                        RAM[state0][time][dec] = bit0;
                                        RAM[state1][time][dec] = bit1;

                                }
                                fprintf (fp2,"%d",outr[L-blk_time-1]);  /* output decoded bits */
                                fflush(fp2);
                                bit_no+ + ;                         /* increment bit counter */
                }
        }
}


/***************************** METRIC COMPUTER ******************************/
/*                                                                          */
/*  This subroutine computes the branch metrics from the "n" received      */
/*  symbols.                                                                */
/*                                                                          */
/****************************************************************************/

metric comp(recsym)
int *recsym;
{
        int symbol_no;                  /* loop counter */
        int sum_recsym;                 /* maximum branch metric */
        int encoded_bit;                /* one bit of branch label */
        int mag_recsym;                 /* received symbol magnitude */
        int sign_recsym;                /* sign of received symbol */

        sum_recsym = 0;                     /* set branch metric to zero */
        p = 0;

        for (symbol_no = 0; symbol_no < n; symbol_no+ + ) {
                mag_recsym = recsym[symbol_no] & 0177;          /* mask the first eight bits */
                sign_recsym = (recsym[symbol_no] > > 7) & 01;          /* extract sign bit */
                encoded_bit = (LABEL[butt] > > symbol_no) & 01;          /* strip label bits */
                sum_recsym + = mag_recsym;      /* sum all the received symbol magnitudes */
                if ((encoded_bit ^ sign_recsym) = = 01) p + = mag_recsym;
        }
        q = (sum_recsym -p);
}

/************************ ADD, COMPARE, AND SELECT ***********************/
/*                                                                        */
/*  Add branch metrics to accumulated metrics.  The pair of sums at each  */
/*  of the states is compared and the smallest is selected.  The output   */
/*  of each of these decisions is the smallest accumulated metric at each */
/*  state and the decision bits which are sent to the traceback memory.   */
/*                                                                        */
/**************************************************************************/
```

```
add_comp_select( )

{
        int s00, s10, s01, s11;    /* the accumulated metrics */

/* add branch metric to accumulated metric */

        s00 = (mi0 + p);
        s10 = (mi1 + q);
        s01 = (mi0 + q);
        s11 = (mi1 + p);

/* determine smallest metric for present two states of butterfly */

        if (s00 < s10) { bit0 = 0; mj0 = s00; }
        else      { bit0 = 1; mj0 = s10; }

        if (s01 < s11) { bit1 = 0; mj1 = s01; }
        else      { bit1 = 1; mj1 = s11; }

/* check constraint length and set output accumulated metrics respectively */

        if (k < 15) mj1 = mj0;

/* determine if accumulated metrics must be renormalized, if so, set flag */

        if (butt = = 0 && (mj0 > > 14) = = 3) flag = 1;
}

/******************************** ENCODER ********************************/
/*                                                                       */
/*  This subroutine generates the labels for each butterfly by utilizing  */
/*  the appropriate generating polynomials.                              */
/*                                                                       */
/*************************************************************************/

encoder( )
{
        int butt;                 /* loop counter */
        int symbol_no;                  /* loop counter */
        int encoded[6];                 /* encoded symbols */
        unsigned int masked;            /* the masked state */

/* encode butterfly labels and do appropriate shifting */

        for (butt = 0; butt < NB; butt+ + ) {
                for (symbol_no = 0; symbol_no < n; symbol_no+ + ) {
                                encoded[symbol_no] = 0;
                                masked = (butt < < 1) & GP[symbol_no];   /* mask the butterfly */
                                for ( ; masked > 0; masked > > = 1)
                                        encoded[symbol_no] ^= masked;   /* sum the bits of butterfly */
                }
                LABEL[butt] = 0;
                for (symbol_no = 0; symbol_no < n; symbol_no+ + )
                                LABEL[butt] |= (encoded[symbol_no]&01) < < symbol_no;
        }
}
```